Issues and Tradeoffs in Document Preparation Systems

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, NJ 07974, USA

ABSTRACT: Users of document preparation systems must balance how much effort they put into producing their documents against how close their output is to what they want. The evolution of document preparation systems is a history of how users and implementers have dealt with this tradeoff as technology improves and as the user population itself evolves.

1 Problem: What You Get is Less Than What You Want

Computerized document preparation systems are not yet forty years old. People, on the other hand, have some thousands of years of accumulated tradition about how documents should be printed to satisfy utilitarian and esthetic criteria. With that much history, there is a rich collection of examples, and people are just as prolific at inventing new styles now as they ever were, so there seems to be no limit to the expectations for document preparation systems.

Furthermore, every user of a document preparation system is an "expert" in what his or her document should look like. The output is visible, so it can be readily compared against some mental or physical ideal, and the shortcomings will be immediately apparent. It is much harder to satisfy the casual user of a document preparation system than, for example, the casual user of a compiler.

Since the user's ideal of perfect output with no work cannot be reached, there is always a tradeoff. How much time or effort or money is the user willing to put into learning or using or buying a system? How much is the user willing to compromise output quality so as to spend less effort or time or money?

The history of document preparation systems is a history of how people have dealt with this tradeoff — balancing the expense and effort of production against the suitability of the output — as changes occur in hardware and software technology, costs, user expectations, the user population, and our understanding of what we are doing.

In the rest of this paper, I will discuss some of these tradeoffs. This is likely to seem like revisionist history; I am not trying to be complete, but only to give a sense of how this notion has affected the growth of our particular field. I am also likely to be provincial, if not parochial, in the choice of topics and examples,

especially by focusing on what might be called the "academic" side of document preparation, systems like TROFF, TEX, and Scribe. The positive side of this is that by speaking on topics where I have first or second hand experience, I hope to reflect more accurately on what happened and why. (An excellent survey of document preparation systems up to 1981 can be found in [Furuta 1982].)

2 In The Beginning

In the late 1950's and early 1960's, interactive computing was in its infancy. When Jerry Saltzer wrote RUNOFF, the standard document preparation system of the day was a secretary with a typewriter. The great attraction of RUNOFF and programs like it was the ability to produce a clean new version of a document after a few changes had been made without having to type the whole thing over. This advantage was and is so compelling that people are willing to suffer severe hardship in preparation and ugly output to obtain it. Of course, when the output medium was just a typewriter, and the formatting program could be controlled with a handful of obvious commands like "leave a space" or "set the line length," the usage was easy and the output entirely adequate.

Programs that come directly from RUNOFF, of which the TROFF family is the most widespread, share a common heritage. They were originally intended to drive typewriter-like devices; the more elaborate mechanisms needed for typesetting were grafted on. And they were designed by pragmatic computer programmers, not typographers or typists; their syntax and semantic model reflect a programmer's view of the world.

Around the same time, the first electronic typesetting machines were developed. These remained largely tools of professional typographers, although programmers were needed to write programs that would control them. The proprietary typesetting programs that come from this period, for example PAGE-1 [Pierson 1972], seem to share the characteristic that they are linguistically even more irregular than the ones developed by non-typographers. Few have been described in the literature.

3 Affordable Typesetters

In 1972, the late Joe Ossanna, who had been interested in text formatting since the advent of RUNOFF, acquired for the Computing Science Research Center at Bell Laboratories a typesetter made by Graphic Systems, for the modest sum of about $12,000. This was the first typesetter that had a reasonable set of features at a price that a small group could justify for purely experimental purposes. The great novelty, at least as far as Graphic Systems was concerned, was that the machine was connected directly to a computer, instead of reading its input from paper tape.

This typesetter became the target of Ossanna's TROFF formatter. Much maligned, but still very much with us, TROFF was the base on which a remarkable number of document preparation tools were built. Technically, TROFF was not a lot different from a commercial system like PAGE-1, although its syntax and style were unrelated. TROFF shared with RUNOFF the basic notion that users decorated their input with commands that would control the formatting process. The richer set of operations permitted by the typesetter meant that a richer set of commands was needed, including in-line sequences for size and font changes, local motions, and non-ASCII characters. In addition, certain operations were sufficiently complicated that ordinary users were not expected to use them directly. Thus Ossanna provided a macro processor with some programming capability so that complicated sequences of operations could be defined and then used without knowledge of their details. Page makeup, for example, was not built in, but was handled by a "trap" mechanism that caused a specified macro to be invoked when a specified amount of output text had been produced. This made it possible to program any desired form of page layout, but only the most dedicated TROFF programmers could do this. This led Mike Lesk to create `ms`, a package of macros that handled the standard page formats in use at Bell Labs. Both `ms` and the much larger `mm` that followed are still widely used, in part because it is easier to put up with their shortcomings or make small changes and additions to them than to write one's own package from scratch.

The combination of TROFF and a typesetter that could print Greek letters, mathematical symbols, and a variety of sizes and fonts was tantalizing in a group that included a fair number of mathematicians. It was clearly possible to print mathematics, but it required heroic measures to force TROFF through the motions, and no one had the stamina for any but the most trivial examples.

How could mathematics be handled more conveniently? One might hope to modify TROFF, but this was impractical. Ossanna took a strongly proprietary view of the program, although he was always willing to add features for his users. The program was at that time written in assembly language, which made it hard to work with. And it was already dangerously close to the maximum program size on the PDP-11; it could not have grown much anyway.

So Lorinda Cherry and I were forced into a good idea: instead of modifying TROFF, we designed a separate program, EQN, that translated a separate language into TROFF commands. Fortunately, this coincided with the early flowering of the Unix operating system, including the notion of pipes, and with the creation of the YACC parser-generator. YACC encouraged a regular grammatical structure and made implementation easy, and the pipeline

```
eqn | troff
```

was completely natural.

It was only a short time later than Mike Lesk took the same idea, though with a very different syntax, and wrote TBL, a language for typesetting tables. Since then, there have been many other preprocessor programs, as people have attacked specialized areas of document preparation.

The TROFF model, with or without preprocessors, remains "open-loop control." The document contains commands that are meant to control its appearance. The commands are changed until the appearance is close enough or the user gives up. The specialized languages help because they make some parts of this much easier to control, but they are still based on embedded commands where the only way to be sure of the effect is to try it.

For in-house use the combination of a standard macro package and some preprocessors proved to be extremely effective, since the only competition was traditional typewriters, perhaps with a special symbol font. Output from TROFF was faster and much better looking, and many people were willing to pay the price of learning the system. (Unexpectedly, one of the greatest advantages of typeset output was that it was compact; judicious use of small sizes and two-column layouts produced legible output in half the space of typewritten text.)

For external use — conference and journal papers, for instance — the case was less clear. From the standpoint of professional typographers, the output quality was inferior, although good enough that authors were occasionally asked if their papers had been previously published. One group of professionals, however, did become interested, because external economics were changing too. The American Physical Society, which at the time used mechanical typewriters to produce 30,000 journal pages per year of complicated mathematics, was searching for a way to reduce their costs. They were willing at least to consider lower-quality appearance in return for lower costs. In 1975 Mike Lesk and I arranged an experiment in which a typist at Bell Labs typed five papers using TROFF, EQN, and TBL; the APS already had measurements of how long their typists had taken. To their surprise, and our delight, the Unix-based system was nearly $2\frac{1}{2}$ times faster. It took many further discussions, experiments, and software fixes, but the APS today produces more than 50,000 pages per year with fewer typists than they used to have. (The AT&T Technical Journal switched from a commercial printer to TROFF in mid-1989. "A prophet is not without honour, save in his own country, and in his own house." [Matthew *XIII*, v. 57])

TROFF was used for book publication as early as 1974, for *The Elements of Programming Style*. Although commercial printers did (and still do) produce better output than any "academic" document preparation system, the process of re-keyboarding into galleys was (and still is) painfully slow and prone to errors.

Since one of the messages of the book was the importance of presenting error-free computer programs, it was mandatory to avoid the errors caused by ordinary composition. The *Style* book was typeset directly from machine-readable source; the programs in it contained no errors from typesetting (although there were a couple of the traditional kind). The book went from conception to camera-ready form in six months and was published two months later. The whole process was so much faster than the norm that almost a year after the book appeared, it was still possible to be told by the publisher's representatives that no such book existed.

4 The Search for Quality

Certainly the major achievement in document preparation systems of the late 1970's and 1980's is Don Knuth's TEX and the web of programs that support it. The high value that Knuth places on the visual appearance of his books is the origin and continuing touchstone of TEX. "The genesis of TEX probably took place on 1 February 1977, when I first chanced to see the output of a high-resolution typesetting machine. [...] I could see no difference between the digital type and 'real' type." [Knuth 1989].

TEX carries the notion of embedded controls far beyond anything envisaged in TROFF. The TEX language has more than 250 basic commands and at least 60 parameters that can be interrogated and set; this is about three times as many commands and parameters as TROFF has. The language syntax is more regular and includes notions of nesting and scope.

TEX makes up pages by combining "boxes" containing individual characters into larger boxes; boxes are positioned relative to each other by adjustable spacing called "glue." Formatting decisions (where to break words into lines, where to break lines into pages, etc.) are expressed in a "penalty function." Minimizing this function by dynamic programming yields the set of breaks that best satisfies conflicting criteria. The boxes and glue model is one of several innovations in TEX; it unifies a number of apparently unrelated processes in a single mechanism. Horizontal justification, vertical justification, and centering, for example, can all be done by adjusting glue.

One sign that the model is not perfect is the number of cases where a box, once built, has to be taken apart again. This shows up in hyphenation and footnote processing. (The original version of TEX strongly discouraged footnotes, and would not split a footnote across a page boundary. Unfortunately, authors want footnotes to work, so the current TEX handles them properly.) Objects that are naturally larger than a page, like program listings and giant tables, are also not gracefully handled.

The elegant dynamic programming algorithm makes it hard to nail down the contents of an individual line, since some change later on in a paragraph can ripple back to the current line. Most typesetting systems set text a line at a time with a greedy algorithm; once the line has been set, future text has no effect on it. The penalty function minimization performed by TEX, however, can change an entire paragraph when a small change is made anywhere. The result is a better layout according to the rules, but in a production environment it is essential to have a convenient way to make changes that have only a local effect. More generally, any document preparation system needs ways to defeat automatic mechanisms, no matter how well-intentioned.

Knuth's work on TEX has had several interesting spinoffs. The perceived inadequacy of existing character sets led to the development of METAFONT, with which families of stylistically related characters can be created. METAFONT has not caught on to nearly the same degree as TEX has, perhaps because there is less compelling a need for new character shapes, and perhaps because it is very difficult to create characters with the right appearance.

What of TEX's facilities for handling special kinds of text processing? The experience with TROFF preprocessors showed the value of separate languages, each focused on and especially suited to a particular area. Separate programs are easier to write and maintain, and it is much easier for anyone with a good idea to contribute a new program than to modify an existing one. There are drawbacks too, however, notably the one-way flow of information, at least in the pipeline model.

TEX takes a monolithic approach: processing of mathematics and, to a lesser degree, tables is integrated into the language. In general, the more integrated a component, the more it will be required to match the rest of its environment. Thus mathematical constructions follow the basic syntax of the rest of TEX; keywords are all prefixed with \, although there are special rules about token parsing, the significance of blanks and newlines, and the use of certain commands and characters. As a result, TEX mathematics looks more cluttered than EQN, but it is more regular. As a simple example, consider this continued fraction:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3}}} \tag{1}$$

in EQN:

```
a sub 0 + 1 over {a sub 1 + 1 over {a sub 2 + 1 over a sub 3}}
```

and in TEX:

```
a_0+{1\over\displaystyle a_1+
     {\strut 1\over\displaystyle a_2+
         {\strut 1\over a_3}}}
```

The commands `\strut` and `\displaystyle` force sizes and spacings to remain constant instead of shrinking automatically.

Of course this comparison is unfair; it happens to be one where EQN's choices of defaults is closer to "right." EQN does not shrink fraction numerators and denominators automatically, nor does it generally take account of the different heights of characters; TEX does both. To over-simplify, EQN tries to make the easy cases easy, and provides relatively few mechanisms for tuning, except for explicit local motions. TEX provides many more parameters that can be adjusted to produce a better appearance.

One of the major failings of TROFF is the degree to which the supposedly abstract notion of a formatting language has been compromised by irrelevant details of the output device it was originally aimed at, and also the idiosyncrasies of the first machine it was implemented on. As Edsger Dijkstra said [Dijkstra 1976], "It is not unusual to find anomalies in existing machines truthfully reflected in programming languages." Dijkstra's remark was made in the context of programming languages, but it is just as valid for formatters. Most document preparation systems are biased by the output device or devices that the builder is familiar with. The output device model can have an inordinate effect on everything else. As an egregious example, the point sizes on a long-vanished mechanical typesetter are enshrined in the input syntax of TROFF. (I pass over the syntax limitations, such as two-character command names, that are a legacy from the original implementation on a PDP-11.)

TEX has similar problems. The most significant is perhaps the fact that TEX believes that fonts are generated by METAFONT, and this has made it complicated to use TEX effectively on most commercial typesetters. For instance, one of the reasons why TEX sets mathematics so well is that it defines what characters are available and it knows accurately what their critical dimensions are. This is a by-product of using METAFONT. Unfortunately, few devices support the model that TEX requires. On most typesetters, fonts, especially math fonts, are a wild collection of ill-assorted characters, and few typesetters support the math characters that TEX assumes it can use. The internal representation of characters is often carefully protected proprietary information. It is difficult or even impossible to create new characters, and it is rare to have any more information about letter shape than some nominal set width. As a result, quality mathematical typesetting requires a lot of tuning, and sometimes legal negotiations with the manufacturer of one's typesetter. The job can be done if the environment is stable enough and the

effort is deemed worthwhile; as mentioned above, the American Physical Society has been publishing with TROFF, EQN, and TBL for many years. More recently, the American Mathematical Society, which has used TEX for some years, switched from Knuth's Computer Modern font to fonts from a typesetter manufacturer; their experience is described in [Youngen 1989].

The graphics model in TROFF is primitive (lines, arcs, circles, ellipses, splines) but richer than the non-existent one in TEX. The LATEX notion of doing graphics by line segments at funny angles is a woefully inadequate substitute. It makes great demands on scarce memory, and it is a terrible input language for people. Neither TEX nor TROFF can do the kinds of graphical transformations of the Postscript model, not even something as simple as setting type at a 90 degree angle for graph labels, (unless a rotated font is used). Part of the reason is that most graphics operations were beyond the capabilities of typesetters when these languages were designed; it is also true that there was no suitable standard for graphics.

5 Affordable Near-Typesetters

The most significant change in document preparation systems in the 1980's has been the development of the affordable laser printer. Everyone can have an output device that, although not of typesetter quality, is superior to anything else.

The first widely available laser printers provided reasonable resolution and speed at a reasonable price, about the same as a typesetter a decade earlier. It is doubtful that TEX would be as successful as it has been if it had not been for the development of such printers. TEX was originally aimed at the Alphatype typesetter, an amazing but slow mechanical wonder. Early non-commercial laser printers were used as proofing devices, and for technical reports. But it was only with the broad availability of the laser printer that people began to take advantage of the capabilities of formatters, especially TEX.

Today, output technology is laser-printer based, but the important change is the emergence of Postscript as a *de facto* standard language for printers and even typesetters. Postscript provides a much richer graphics model than did any typesetter of the past, with arbitrary curves and transformations, area filling, clipping, half-toning, and the like. It also provides a different model of character generation from the linear scaling of early typesetters and the "everything is just a bitmap" view of TEX and METAFONT.

There is no integrated way to get at these richer facilities from within TEX, and only the most primitive way in bare TROFF. The current trend is electronic cut-and-paste, the inclusion of separately prepared Postscript in holes left by formatting commands. But this does not handle, for example, pictures that contain mathemat-

ical expressions. Another option is to use what amounts to an assembly-language window in the formatting language: `\special` in TEX, `\X'...'` in TROFF. Such escapes, though necessary, have all the undesirable properties of assembly language coding. TROFF preprocessors like PIC and IDEAL, and similar ones for TEX (e.g., [Olejniczak 1989]) provide somewhat more integrated access to graphics, but they too have only limited graphics capabilities. But user expectations for graphics are now far beyond the descriptive capabilities of these formatting languages, even when augmented with special languages.

6 User Interfaces: What Does the User Have to Face?

The user interface for the major document preparation systems remains a batch system, a form of remote control or action-at-a-distance. Furthermore, at the level of the built-in primitive operations of these programs, one has to be a programmer of considerable sophistication and experience to control them.

How does one arrange an interface so that users need not be programmers to get something done? And how does one arrange that users need not be programmers to accomplish something different from what the designer allowed for?

The Scribe formatter was the first serious attempt to address this problem. The idea was to specify a document in terms of its logical components — title, abstract, paragraph, list of items, reference, quotation — and have those logical components interpreted according to a style sheet that caused them to be printed with the desired appearance. A family of style sheets permits much the same input to be printed in a variety of physical appearances; in addition there are facilities for making modest changes of style parameters. This notion was implicit in Mike Lesk's `ms` macro package for TROFF, but it was not stated very explicitly, and it was never enforced: it was always possible to escape to TROFF. (The MONK formatter [Murrel 1989] provides a Scribe-like TROFF interface.) Standards like SGML and ODA (see, for example, [Brown 1989]) are meant to support logical descriptions of documents.

LATEX, by Leslie Lamport, is a macro package for TEX. Like Scribe, it strongly encourages a logical model of use, but it permits easy and even inadvertent access to the underlying TEX commands, which users are advised to use cautiously.

LATEX and `ms` suffer from the fact that the extension mechanism in the underlying formatter is a macro processor. Unfortunately, macro processors, although the easiest extension languages to build, are also the worst of programming languages. A trivial (but perpetual) problem is how to protect embedded commands against premature evaluation. In TROFF, the symptom is a plethora of additional \'s. In LATEX it is a "fragile" command, which is a command that can fail to work because of unwanted evaluation of a macro argument. The remedy is to

have the user add `\protect` commands at judicious points. These are exactly the equivalent of adding an extra level of backslashes in TROFF, and are about as hard to understand and get right. Error recovery in a macro environment is difficult as well: everything is valid input, so if something goes wrong, commands may be converted into text and vice versa.

As an aside, one of the most challenging tests of a formatter's programmability is to simulate the format of APS journals, which affect an alternation of one- and two-column material on a single page, apparently to pack in as much text as possible. Mike Lesk wrote a TROFF macro package for APS format in the mid-1970's; a recent somewhat less complete version for TEX is described in [Benson 1989]. (The APS still does page layout by hand.)

Postscript is an interesting advance. Postscript recognizes the need for a language as an extension mechanism, and supports the elegant idea of sending a program instead of just data. The Postscript language is stack-based; operands are pushed onto and popped off a stack by operators. Linguistically, stack languages are probably the second-easiest extension mechanism, and are an improvement on macro languages. But they suffer almost as much from being hard to understand, and are just as prone to disintegration when something goes wrong. (See, for instance, [Gonczarowski 1989].) The problem in Postscript is not so much that data turns into program, but that the wrong kind of data appears on the stack. Because Postscript associates types with data, errors can often be caught quickly, but recovery remains difficult.

Even if a language starts out declarative, there is an inexorable pressure to add programmable features to it. For example, PIC was originally rather simple, but people wanted loops and conditionals; when they were added to PIC, people wanted them in GRAP as well.

Whether or not the basic tool is programmable, there is always a need for escape, because people want to do things that are simply not possible in the underlying process. For example, TROFF includes a command `.sy` that executes any operating system command; output from the command can be collected and inserted into the document if desired. This provides a trivial way to do "active documents." Its use is routine for complicated formatting processes, and it is often used to fetch text like program listings that must be processed before being formatted. (The `.sy` command is a potential security problem — an unscrupulous person could send a document that does something untoward when the victim runs it through TROFF. It's easy enough to defend against this if one plans ahead, but few would.)

There is no corresponding mechanism in TEX; if one wants to perform other operations as part of formatting, they have to be done before (in some kind of preprocessing) or after (in the DVI processor, for example).

Both TROFF and TEX have mechanisms that cause some piece of text to be passed to the output uninterpreted. TROFF's `\X` is used mainly for *ad hoc* extensions like sending small amounts of embedded Postscript, although it is also used extensively by the PM page makeup postprocessor [Kernighan & Van Wyk 1989]. TEX's `\special` is the basis of several tools, particularly graphics processors. In Postscript, job and page control features that don't fit into the primitives of the language are added through structured comments [Adobe 1985]. The problem with all such mechanisms is precisely their *ad hoc* nature — there are no standards for what gets passed through and thus such documents tend not to be portable.

Macro packages like LATEX or `ms` are valuable for encapsulating complicated sequences of commands, but they are limited to the syntax rules of the base language, and they cannot do more than what the base language can express. The former may be limiting to the user; the latter is limiting to the implementer.

Cooperating but separate processors provide an alternative that avoids these problems, at the price of some others. The family of TROFF processors is the most notable example; these separated components can be combined in ways that weren't thought of in the original design, and new ones can be added as people think of them. The advantages and drawbacks of the preprocessor approach have been spelled out before [Kernighan 1989]. Here I would like to stress a few of the lessons that we have learned.

One is that people will abuse tools in remarkable ways, far beyond anything that the author ever thought of. For example, Figure 1 is a version of the classic map that shows attrition during Napoleon's march to Moscow and his subsequent retreat. Remarkably, this PIC version, 270 lines of source, was created by hand.

If a program accepts textual input, then one can write a program to produce that output. It soon becomes apparent that machine-generated input stresses programs much harder than people do, and in very different ways. For instance, a complicated PIC picture drawn by hand might have a few hundred components; a picture generated by a program (a graph produced by GRAP, perhaps) can easily have 50,000. Fixed-size limits in programs are always too small. Regularity of input syntax is also important for machine-generated input. For example, during the development of GRAP, I discovered to my embarrassment that GRAP was producing numbers in scientific notation and PIC would not accept them.

Another lesson is the often inordinate difficulty of getting the target program to play dumb. For example, in early work getting TROFF to drive other typesetters, I usually found it hard to convince the typesetter-resident formatting program to turn off its own hyphenation and justification, or to place characters exactly where I requested them. This same problem occurs in TEX and TROFF, of course, where it is sometimes hard to force these programs to put output at specific positions.

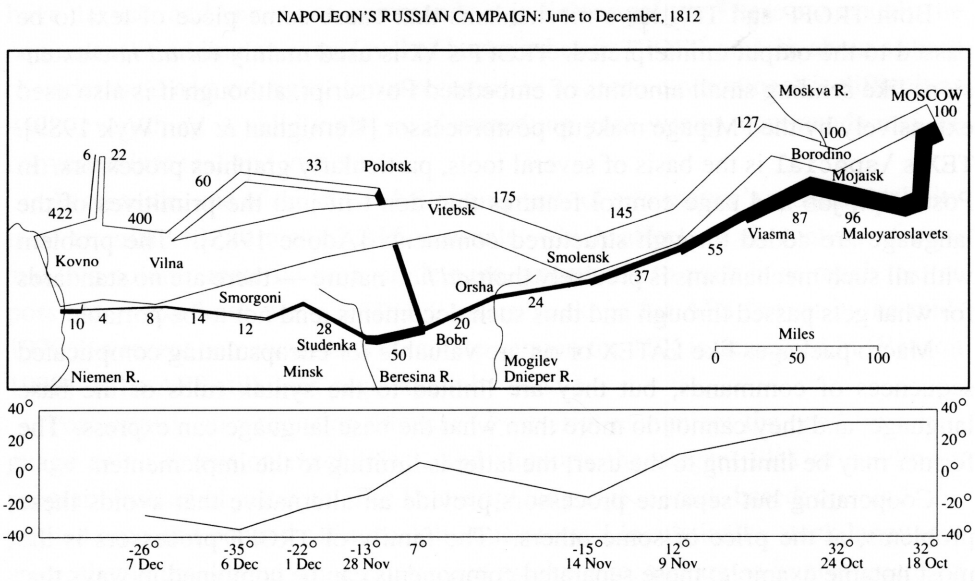

Figure 1: Napoleon Visits Moscow

TROFF, for example, treasures up horizontal and vertical motions and puts them out as late as possible. This makes it difficult to include embedded Postscript, since there is no guarantee that the logical position that TROFF has computed is reflected at the right time in the output that the Postscript postprocessor sees.

Another observation is that, just as people write programs to generate input, they write programs to process output further. Many of the same issues arise here, and some new one. As a trivial instance, it helps to have readable output; TROFF's ASCII output can be processed by any standard tool, while TEX's more compact binary DVI cannot.

Most important, it is necessary to preserve information so that subsequent programs can use it for things not thought of in the original design. As one instance, the family of programs that handle multi-directional text process text an output line at a time. This is not possible in TEX: "Because there are no end-of-line markers in TEX's DVI output format, the system structure adopted in this paper cannot be applied to make a tri-directional version of TEX. Instead one must make modifications to TEX either to have it do the reorganization or to have it emit end-of-line markers [Becker 1989]." On the other hand, even though TROFF output marks line boundaries, it does not reliably include the height and baseline for each line; this oversight leads to inordinately complicated code in PM that attempts to

infer the missing information.

Another lesson is that although error reporting and recovery can often be primitive in such programs, and can always be primitive in the early stages, ultimately reporting and recovery have to be good. One trivial observation: we have had to add to TROFF and each preprocessor a command to reset its internal idea of the line number and source filename so that diagnostics are reported in terms of the user's input, not the result of multiple steps of processing. (The latest revision to TEX adds a new parameter called `\inputlineno` but makes it read-only; this precludes some future TEX preprocessor from setting it to give a true line number.)

7 What You See Is ...

Another major hardware advance of the late 1980's, the widespread availability of bitmap screens, has made possible an entirely new style of formatter. Rather than embed commands and hope for the best, one can interact with a reasonably faithful display of what the document will look like and change it to taste in real time. The appeal is obvious and, for many kinds of documents, a WYSIWYG system is just what people want and need.

There is no gainsaying the advantages, but the disadvantages should also be noted. First, there is a significant sacrifice in facilities; a survey [Walter 1988] of commercial technical publication systems covering the gamut from simple desktop publishing to professional quality indicates that none is really up to the mark. Similar results are noted in a smaller review [Gruman 1987].

It is important to realize that interactive systems are not a panacea. It is frustrating to be forced to interact with a system when all one is doing is telling it the same thing that it has already been told repeatedly in the past. Once a task is well understood, it should be relegated to batch processing. The interface needed by experts may be baffling to novices, but experts can be hobbled by an interface aimed at novices.

It is also hard to augment the set of tools at the user's disposal when the only interface is through an interactive system. Interactive systems are generally closed: an interactive system represents a dead end in the use of a program. If a program has to be run by a person, then that program can not serve as the target of other programs. For example, if TROFF were only an interactive formatter, there would be no PIC language for drawing pictures; similarly, if PIC were an interactive picture editor, there would be no GRAP program for drawing graphs.

There is a strong tendency for interactive systems to concentrate on the visual aspects of preparing documents, at the price of the logical structure; indeed, it may be impossible to capture the logical structure that the user has in mind as the

document is being entered. Leslie Lamport suggests that interactive systems are "good for serious document production only to the extent that they act like logical systems" (that is, that they work in terms of the logical structure of the document, not its visible representation) [Lamport 1988]. Clearly what is needed are systems that combine the best of interactive and batch. This is an active area of research and increasing commercial success, but it remains difficult, and no current system is more than an approximation.

8 Conclusions

What of the future? It is easy to imagine that the future will be more of what we are seeing now: our productivity and the quality of our output will increase as the sophistication of our tools increases, yet we will always face the same kinds of problems and the same tradeoffs between what is possible and what we are willing to pay for.

Many of the same problems still exist. Although our tools for plain black and white two-dimensional typography are better than they were, they remain too hard to use, especially by novices, and as programs are far too hard to extend. We need more work on comprehensible and flexible batch user interfaces.

Large multi-author documents such as manuals for complicated systems present document preparation issues beyond mere formatting: document management, on-line retrieval, electronic distribution, and the like.

There are new areas that will demand new tools and new ways to deal with their inclusion in documents. For example, we are in the earliest stages of widespread use of color. How will we provide access to color in an effective way?

Character sets and fonts will continue to proliferate, and if we hope to have our systems used elsewhere in the world, we will have to learn to handle different and often very large character sets.

How will we cope with "active documents"? Just because a document is active, I may not want to react to it. We will still need specialized languages to deal with particular aspects of such complicated documents: to access data bases, to play music, to show video, to control scripts.

We will continue to need good work in algorithms and data structures, if we are to be able to make our programs work at all, let alone be efficient and easy to evolve. Most of the existing programs are large and complicated. What can we do to make them more manageable?

How will we deal with the continuing tension between standards, whether *de facto* like TEX and Postscript, or *de jure* like SGML and ODA, and the necessity for evolution?

In spite of these puzzles, we will continue to make progress. A decade or two from now, today's problems will seem minor, and today's systems will seem as primitive as those of twenty years ago seem now.

9 Acknowledgements

I am very grateful to John Hobby, Doug McIlroy, Peter Nelson, Rob Pike, Howard Trickey, and Chris Van Wyk for helpful comments on this paper.

References

[Adobe85] Adobe Systems, Inc., *PostScript Language Reference Manual*, Addison-Wesley, 1985, pp. 265ff.

[Becker89] Zeev Becker and Daniel Berry, "TRITROFF, An Adaptation of the Device-Independent TROFF for Formatting Tri-directional Text," *Electronic Publishing*, vol. 2, no. 3, October 1989, pp. 119–142.

[Benson89] Gary Benson, Debi Erpenbeck, and Janet Holmes, *TUGboat*, vol. 10, no. 4, December 1989, pp. 727–742.

[Brown89] Heather Brown, "Standards for Structured Documents," *Computer Journal*, vol. 32, no. 6, December, 1989, pp. 505–514.

[Dijkstra76] Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976, p. 8.

[Furuta82] Richard Furuta, Jeffrey Scofield, and Alan Shaw, *Document Formatting Systems: Survey, Concepts, and Issues Computing Surveys*, vol. 14, no. 3, September 1982, pp. 417–472.

[Gonczarowski89] Jakob Gonczarowski and On G. Paradise, *Electronic Publishing*, vol. 2, no. 3, October 1989, pp. 157–167.

[Gruman87] Galen Gruman, "Software Reviews: Desktop Publishing," *IEEE Software*, vol. 4, no. 3, May 1987, pp. 70–77.

[Kernighan89] Brian W. Kernighan, "The Unix System Document Preparation Tools: A Retrospective," *AT&T Technical Journal*, vol. 68, no. 4, July/August, 1989, pp. 5–20.

[Kernighan & Van Wyk89] Brian W. Kernighan and Christopher J. Van Wyk, "Page Makeup by Postprocessing Text Formatter Output," *Computing Systems*, vol. 2, no. 2, Spring, 1989, pp. 103–132.

[Knuth89] Donald E. Knuth "The Errors of TEX," *Software—Practice & Experience*, vol. 19, no. 7, July 1989, pp. 607–685.

[Lamport88] Leslie Lamport, "Document Production: Visual or Logical?" *TUGboat*, vol. 9, no. 1, April 1988, pp. 8–10.

[Murrel89] Sharon L. Murrel and Thaddeus J. Kowalski, "Monk: A High-level Text Compiler," *AT&T Technical Journal*, vol. 68, no. 4, July/August 1989, pp. 45–60.

[Olejniczak89] Rolf Olejniczak-Burkert, "*texpic*—Design and Implementation of a Picture Graphics Language in TEX à la pic," *TUGboat*, vol. 10, no. 4, December 1989, pp. 627–638.

[Pierson72] John Pierson, *Computer Composition using PAGE-1*, Wiley-Interscience, 1972.

[Walter88] Mark E. Walter, Jr., *Technical Documentation Systems*, Seybold Publications, 1988.

[Youngen89] R. E. Youngen, W. B. Woolf, and D. C. Latterner, "Migration from Computer Modern Fonts to Times Fonts," *TUGboat*, vol. 10, no. 4, December 1989, pp. 513–519.